1 Parser

Once the actual code has been separated from proof code, the Parser then parses into an abstract syntax tree.

```haskell
module Parser where
  import Data.Map (Map)
  import qualified Data.Map.Strict as Map
  import Lexer (lexify)
```

The NativeAST is, for now, a placeholder for whatever type is produced by the lexer/parser of the language being proven.

```haskell
  data NativeAST = NativeASTNode

  parseCode :: String -> AST
  parseCode code = transformAST NativeASTNode

  transformAST :: NativeAST -> AST
  transformAST native = ID "The code" VEmpty
```

Once the code has been turned into a NativeAST, it is then transformed into the AST by the pluggable Transformer. Meanwhile, the proof code must also be converted into the definitions used to prove the program. These are represented by the same AST as the code, but this transformation is handled here.

```haskell
  data AST = Scope [AST] -- DeclList
           | ID String AST -- name ArgumentList
           | ArgumentList [AST] -- [Annotation]
           | TypeOf AST -- Value
           | Annotation String AST -- name Type
           | Let AST AST AST -- ID Type Body
           | Arrow AST AST -- Annotation Type
           | Function AST AST -- ID Body
           | Application AST AST -- Function Value
           | Exists AST AST -- Annotation Body
           | IntroExists AST AST -- Type Value [will this need another
             argument?]
           | ElimExists AST AST -- Exists Body [how does this work again?
              does it need another argument too?]
```

```
| And AST AST -- Type Type
| IntroAnd AST AST -- Left Right
| ElimAndLeft AST AST -- And Body
| ElimAndRight AST AST -- And Body
| Or AST AST -- Type Type
| IntroOrLeft AST AST -- Or Value
| IntroOrRight AST AST -- Or Value
| ElimOr AST AST AST -- Or LeftBody RightBody
| Contradiction
| ElimContradiction AST AST -- Contradiction Body [does this
    have a body? contradiction usually means done]
-- value nodes
| VNatural Int -- Value
| VFloat Float -- Value [is this needed? or just define as a
    pair or in STL]
| VChar Char -- Value [is this needed? or just define as an
    int or in STL]
| VBoolean Bool -- True/False
| VCons AST AST -- Head Tail
| VEmpty -- empty list
| VSymbol String -- For
| VNull -- the empty value
| VUndefined -- the non-existent value
-- induction [do these need that 4th param like last time?]
| IndNatural AST AST AST -- Int BodyS BodyZ
-- [how to use a float? is float usage STL?]
-- [how to use a char? is char usage STL?]
| IndBoolean AST AST AST -- Bool BodyT BodyF
| IndList AST AST AST -- List BodyL BodyE [is this correct?]
-- [how to use a symbol?]
-- [how to use null?]
-- [how to use undefined?]
```

The first step in parsing the proof code is, of course, lexifying it. This step is taken on by the Lexer. After that, we move on to parsing, which uses the parser generated by Happy.

```
parseProofs :: String → AST
parseProofs proofText = parse $ lexify proofText -- TODO: parse will be
     imported from the happy place
```

Once parsing is complete the two trees are merged into one containing the actual code annotated by proof terms. This is the final tree which is returned to the Compiler to be used by the Analyzer in assuring that the program is valid.

```
annotates :: AST → AST → AST
annotates proof code = Annotation code proof
```